View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by VU Research Portal

VU UNIVERSITEIT AMSTERDAM

VU Research Portal

Algorithmic Design of the Globe Location Service - Basic Update Algorithms

Hauck, F.J.; van Steen, M.; Tanenbaum, A.S.

1996

document version
Publisher's PDF, also known as Version of record

Link to publication in VU Research Portal

citation for published version (APA)
Hauck, F. J., van Steen, M., & Tanenbaum, A. S. (1996). *Algorithmic Design of the Globe Location Service - Basic Update Algorithms*. (VU Technical Report; No. IR-413 (December)).

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

E-mail address:
vuresearchportal.ub@vu.nl

Download date: 22. May. 2021

Algorithmic Design of the Globe Location Service

Basic Update Operations

Franz J. Hauck
Maarten van Steen
Andrew S. Tanenbaum

Internal report IR-413
December 3, 1996

Abstract

In Globe, a distributed shared object provides one or more contact addresses to processes that want to bind to the object. Contact addresses are maintained by the Globe location service, and specify where and how an object can be contacted. As such, a contact address is location dependent. Whenever a process wants to bind to an object, it provides the location service with a globally unique, and location-independent object handle. The location service is capable of returning addresses that are nearest to the process that requested them.

In this report, we describe the basic architecture and the algorithmic design of the Globe location service. We concentrate on the algorithms for updating the set of contact addresses, i.e., basic algorithms for inserting and deleting contact addresses. The algorithms are basic in the sense that they do not take optimizations into account. With respect to fault-tolerant behavior, the algorithms discussed in this report can handle failures caused by network partitions, but cannot recover the service from node crashes.

vrije Universiteit

Faculty of Mathematics and Computer Science

1 Introduction

The Globe Object Model provides a notion of distributed shared objects [2, 3]. Distributed shared objects are truly distributed, which means that parts of an object can be distributed across different hosts or address spaces and that each part may contain a fraction of the entire object state. The parts are built using local objects, which may be composed using other local objects (composite objects) but are never distributed.

If a process wants to communicate with a distributed shared object it has to bind to that object. First, it has to find the unique identifier of the object, the object handle. This can be retrieved by using a name server or by other means. Second, the object handle is used to find one of the local objects that provide a contact point and a protocol for communicating with the contact point. Third, a new local object in the address space of the process is created that is able to use the required contact protocol. This local object is initialized with the communication address of the contact point in order to get in touch with the distributed object. The communication address combined with a protocol identifier is called a contact address. The newly created local object is considered to be part of the distributed object.

For this binding process, there is a need for a service which is able to map object handles to contact addresses. We call this service a **location service**. Object handles are location-independent names, whereas contact addresses are not: The latter contain a location-dependent network address. When a contact point moves to a new location it has to update its contact address in the location service. The binding process hides the current location and clients can bind to the object without knowing any location information. If the object is replicated and provides multiple contact points, the location service will return the nearest contact address for a client. Thus, the binding process also hides that an object is replicated.

This report describes the basic architecture and algorithms of a location service for Globe and is organized as follows. Section 2 explains the functionality of the location service in terms of interfaces and semantics. Section 3 introduces the basic architecture of the service and an outline of the algorithms used for inserting, deleting, and looking up contact addresses. In Section 4 we define a syntax for the description of our algorithms. In Section 5, we describe the data structures and some global consistency rules. The basic algorithms are described in Section 6 (insert operations) and in Section 7 (delete operations). The lookup operations are briefly discussed in Section 8, but details are deferred to a forthcoming report.

Each update algorithm is accompanied by an informal correctness proof. Although simulations using the Promela toolkit [1] indicate that our algorithms are indeed correct, we intend to present a more rigorous approach in a forthcoming report. At that point, the update algorithms will be accompanied by algorithms for looking up contact addresses, as well as results from an experimental implementation that we are currently working on [4]. In this light, the material presented here is to be viewed as a status report on our current research into a worldwide location service.

2 Functionality

An **object handle** is a location-independent identifier for an object [2]. Our location service is able to map an object handle to one or several contact addresses of that object. A **contact address** contains all the information to contact an object. In the context of the Globe Object Model, a contact address contains a protocol identifier and a communication address. For the location service, the contents of

a contact address are not relevant, except that the address is related to some location. The location service is able to return the "nearest" contact addresses for an object handle. This is done with respect to the notion of distance within the location service. This kind of locality helps to reduce the network load and meets the intuitive way of communicating to the nearest available part of the object.

The location service is considered to be some sort of a distributed database containing all the contact addresses of registered objects. The external interface of the location service contains three operations:

insert This operation allows to add a contact address for an object handle to the internal database of the location service.

delete This operation removes a contact address from the database.

lookup With this operation, clients can lookup one or more contact addresses for an object handle. This operation prefers near contact addresses.

As there may be race conditions between updates and lookups, there is always an end-to-end check used in the binding process. In Globe, the unique**object identifier**[1] of an object is passed as part of the binding protocol [2]. If a client has bound to a distributed object it will immediately find out whether it is the desired object. If it is not the binding process failed and has to be repeated. Thus, the location service does not take care about these race conditions. Instead, it provides a best effort service in the sense that it tries to keep its database as up to date as possible.

3 Basic architecture

This section introduces the basic architecture of the location service which appears as a special distributed shared object with numerous contact points. This object is special because processes are automatically bound to it.

The internal architecture is based on a regional partition of the location domain. For the Globe system, the location domain is one or more large-scale networks and their address spaces, e.g., the TCP/IP-based Internet. The regions of that partition are relatively small and will probably correspond to local-area networks (e.g., campuses and small organizations or enterprises). These regions are called**basic regions**.

The basic regions are the possible locations of a contact point relative to the location service. In other words, a contact address is not said to be located in a specific network or at a specific host. Instead, contact addresses are said to be located in a specific basic region.

The basic regions are combined, to form larger regions on a second level. The second level regions are further combined to form third level regions and so on. Finally, there is one top-level region that covers the entire network domain. Note that a contact address belongs to exactly one region at each level. A region in one of the various levels may represent a campus, a city, a country, a global enterprise, etc.

Each region at each level is assigned to a**(directory) node** of the location service which is responsible for that region. Figure 1 shows a graphical representation of such a structure. The structure of nodes logically is a tree.

[1]The object identifier is part of the object handle.

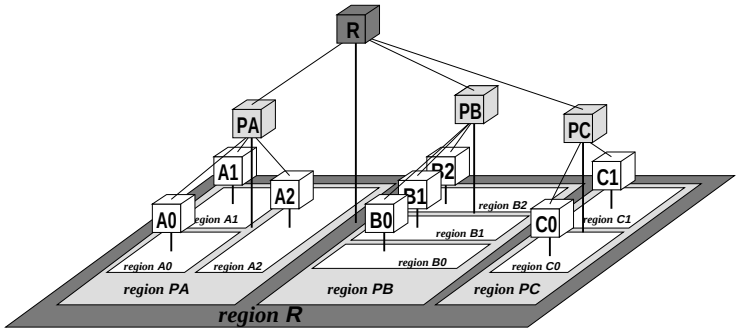

Figure 1: The tree of directory nodes of the location service and its regions.

The location service has no function for mapping contact addresses to basic regions. Instead, a client has to contact a leaf node of the location service for insertion. The region associated to the leaf node is the location of the inserted contact address with respect to the location service. Internally, a contact address gets an additional field which identifies the basic region of insertion. This will allow us to immediately divert a delete operation from any leaf node to the leaf node where the original insertion took place, thus avoiding a look up operation to find where the to–be–deleted address is currently stored. We call the combined data a **regionalized contact address**. This scheme allows the location service to consider the raw contact address as opaque data.

To contact the location service, each client needs a **location resolver** as a mediator. In Globe, clients are always bound to a location resolver which itself is bound to the leaf node of the basic region in which the client resides. The location resolver provides the interface described in Section 2. The resolver regionalizes contact addresses on insert and delete operations. As a side effect, the deletion of a contact address has to take place at the same leaf node as its insertion, because a comparison of regionalized contact addresses includes the basic region of insertion.

3.1 Data storage

For a particular object handle, all contact addresses and internal data structures are stored in **contact records**. For each object handle, there is at most one contact record per node of the location service. In the rest of this report we consider only the data structures for exactly one particular object handle. We also assume that each node has a contact record for that object. This simplifies our presentations and the descriptions of the algorithms. In a final implementation, each operation needs an additional parameter to select the particular object handle to be updated or looked up; each node needs to store a set of contact records, one for each object handle.

If we insert the first contact address for an object handle, the address is usually stored in the corresponding leaf node as part of the contact record for this object handle. Additionally, the directory nodes of every region where the basic region is part of (the parent nodes in the tree) store a **forwarding pointer** to its immediate child.

Figure 2 shows the tree of directory nodes and the contact records for an object handle with one inserted contact address. If there are multiple contact addresses for an object then there can be multiple leaf nodes to store them. For each of them, there is a path of forwarding pointers from the root node to the leaf node, but there is only one forwarding pointer between two nodes (see Figure 3). A contact record

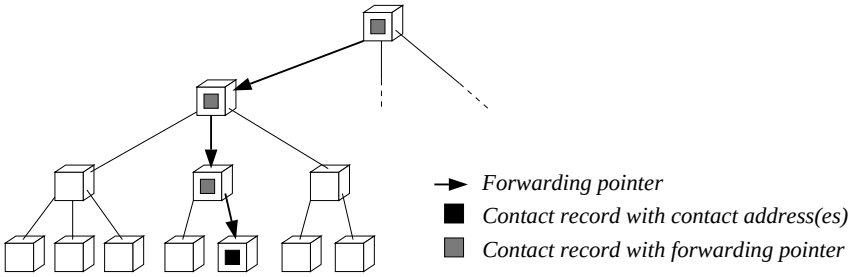

Figure 2: Part of the node tree with one contact address.

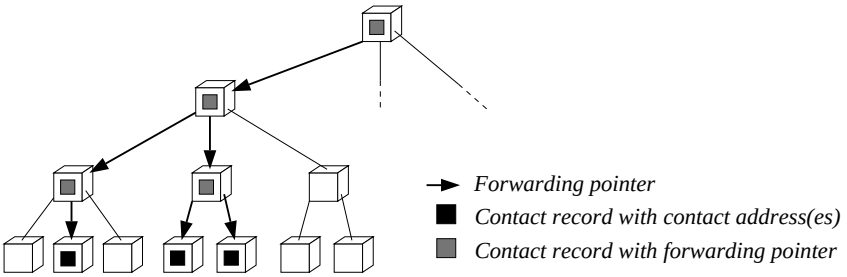

Figure 3: Possible contact records for multiple contact addresses per object handle.

can also store multiple contact addresses if they happen to be located in the same region.

The lookup operation starts in the leaf node of the basic region where the client resides. If it does not find any contact addresses there, it forwards the request to its parent node, thus broadening the search to a larger region (see Figure 4). If it finds a forwarding pointer, the operation will follow that pointer to the stored contact addresses. If it does not find anything it will repeat forwarding the request to its parent node. Finally, at the root node there will be a forwarding pointer or there is no address at all in the location service. This procedure clearly prefers local addresses. The tree structure induces a notion of distance between regions which may be neither the geographical distance nor the round-trip time distance between two hosts. Obviously, basic regions that are topologically close but belong to different immediate subregions of the root node may still be treated as being distant. This is inherent to all strict hierarchical organizations of geographical regions. Future versions of our location service will improve the behavior of algorithms on this point.

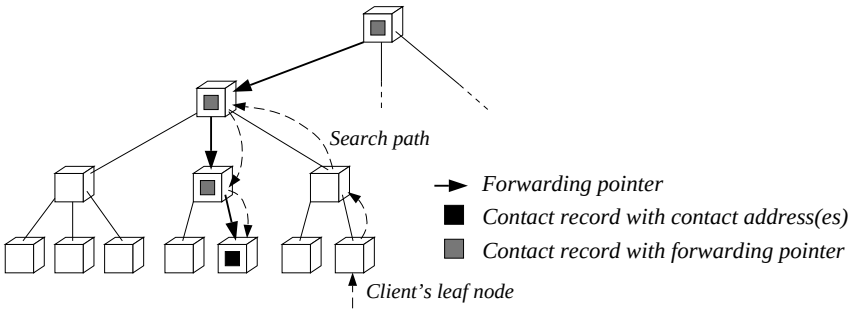

Figure 4: The search path of a lookup operation.

To reduce the search path, we exploit pointer caches at each node. The caches store pointers to nodes where contact addresses can be found. The pointer caches in all the nodes along the search path are filled and updated on each lookup operation. A cached pointer is immediately invalidated if it does not point to a node storing contact addresses. Additionally, cached pointers expire using some timing mechanism if they are not used and thus validated. Caches are not further discussed in this report.

3.2 Higher level storage

If a contact point frequently moves within a certain area it has to often update its contact address. This is done by inserting a new address and deleting the old one. If this does not happen within a single basic region, but instead, involves multiple basic regions, there may be a lot of overhead for establishing and deleting forwarding pointers. Besides that, cached pointers of lookup operations are more likely to be stale.

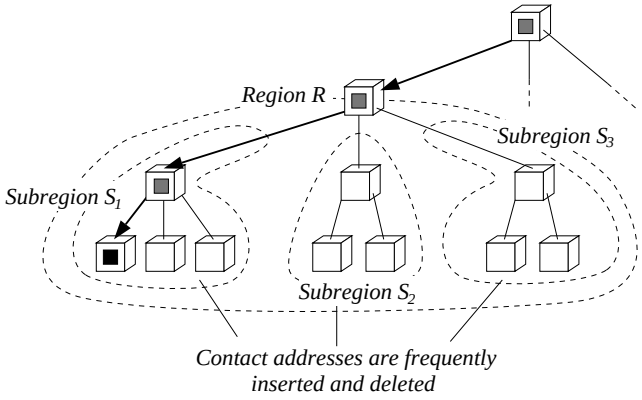

Figure 5: Snapshot of the data structures of a very mobile contact point moving among regions S_1, S_2, and S_3.

Figure 5 shows the data structure of an object with a contact point that is frequently moving within the regions S_1, S_2, and S_3. The node representing region R can detect this situation by accumulating history information about the changes of forwarding pointers that have to be frequently deleted and re-established in the contact record. In cases such as this one, the directory node of R may decide to disallow storing the contact addresses in the leaf nodes, and instead, to store them itself.

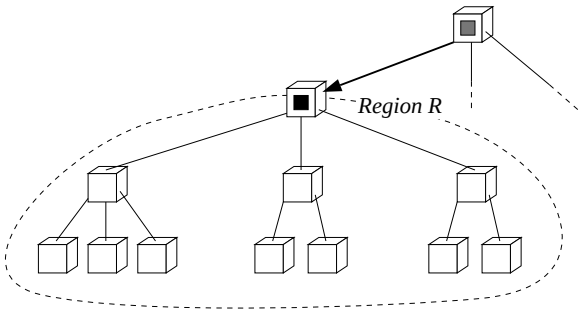

Figure 6: Higher level placement of contact addresses of a very mobile contact point.

Figure 6 shows this situation. Now, the contact address stored in the node for region R may change

regularly because the contact point moves within R, but the *place* where the address is stored is stable. There is no need to update any forwarding pointers. Cached pointers do not become stale as long as the contact point remains moving within region R.

As soon as the contact point remains stable for a reasonable period of time in one of the regions S_i, the contact address is again pushed down to the corresponding leaf node. In this report, we do not consider how stable storage places are determined.

3.3 Partitioning of nodes

For reasons of scalability, each node of the tree has to be partitioned into **subnodes**. This partitioning is done on the basis of the object-handle space such that each subnode is responsible for a disjoint set of object handles. In this report, we focus on the algorithms of the location service. As the partitioning does not affect these algorithms we do not discuss partitioning in further detail, and refer the interested reader to [5].

4 Syntax

In the following sections, we introduce a pseudo programming language to describe the data structures and algorithms of our location service. In all cases, comments are placed after em-dashes and last to the end of the current line:

```
— This is a comment
```

Furthermore, variable definitions are given using the **let** keyword:

```
let x : Integer;
let x : Integer = 4;
let x := 4;
```

In the first line, a variable x of type Integer is defined. The second line additionally initializes the variable with some value. In the third line, the type of the variable is derived from the type of the initializing value.

4.1 Communication

The nodes of the location service communicate with each other using an RPC mechanism. In the syntax, the RPC is represented at the client side by the **call** primitive.

```
let response := call insert_rca at parent;
```

This piece of code invokes the operation insert_rca at the node denoted by parent (of type NodeID). The variable response is defined and initialized with the received response. The **call** primitive is blocking until a response arrives.

The RPC semantics is *exactly-once.* This means that we assume we have an implementation at hand that ensures that the call is carried out exactly once at the callee's side. In general, this is a false assumption in the presence of node crashes. However, as we deal only with network partitions, we can safely assume exactly-once semantics for the time being. In addition, we expect that we need to change our reasoning in only minor details when taking node crashes into account as well, as all our operations are idempotent.

At the server side, the keyword **caller** denotes the node identifier (of type NodeID) of the sender of an RPC. It equals NIL if the sender is a location resolver.

4.2 Concurrency control

Within one node, operations are serialized and invoked in the same order as sent by a client. Operations sent by different clients are invoked in an arbitrary order, but are still serialized.

While an operation is blocked on a **call** primitive, thus waiting for a subsequent RPC to finish, other operations may be scheduled. If multiple operations wait on a **call** primitive then they are unblocked in the order of **return** statements occured in the server, or phrased differently, they are scheduled in the order of response messages sent by the same server node. However, we always require that there is only one operation active at a time per node. Note, that we assumed that there is only one object handle. In a final implementation there can be full concurrency between operations concerning different object handles.

How the semantics for communication and concurrency control are implemented is described in [4].

4.3 Views

Our specification of the various algorithms for inserting, deleting, and finding contact addresses makes use of views and view series. A **view series** is an appearance of the value of a variable that need not coincide with the actual value. The variable itself, however, is left unaffected by a view series. A **view series** is associated to a variable and contains a series of views of that variable. If we have a variable *x* then

```
let vseries : view series of x;
```

defines a view series for that variable. The view series can be evaluated like a read-only variable. Initially, its value equals the value of the associated variable.

There are a number of basic operations defined for a view series: append, apply, and remove. The **append** operation appends a new view to the view series. Views are given as **view expressions** which have to be applied to the original value of the variable in order to get its viewed value.

```
let x : Integer = 4;
let vseries : view series of x;
append view {self + 1} to vseries;          — Append Operation
append view {self * 2} to vseries;          — Append Operation
— vseries = 10; x = 4
```

The pseudo-variable **self** points to the variable for which the corresponding view series is defined. However, note that it is wrong to *substitute* that variable's present value in a view expression. In the

example, the viewed value of vseries is always $2*(x+1)$ regardless how x is changed. The value of a view series (or, likewise, the value of **self** in a particular view) is obtained by taking the actual value of the associated variable and evaluating all (preceeding) view expressions of the series in the order that they were appended. We say that the variable *appears* to have its viewed value.

The least recent view appended to a view series, i.e., the first in the series, can be applied to the associated variable using the **apply** operation. In this case the associated variable is changed and the view is removed from the series.

```
let x : Integer = 4;
let vseries : view series of x;
append view {self + 1} to vseries;
append view {self * 2} to vseries;
apply view to vseries;                                  — Apply Operation
— vseries = 10; x = 5
```

The least recent view can also be removed without applying it to the associated variable using the **remove** operation. Thus, the effect of the view is undone.

```
let x : Integer = 4;
let vseries : view series of x;
append view {self + 1} to vseries;
append view {self * 2} to vseries;
remove view from vseries;                               — Remove Operation
— vseries = 8; x = 4
```

The built-in function **views** retrieves the length of a view series.

```
let nr := views (vseries);
```

4.4 Data structures

For describing our algorithms, we make use of sets which may also be indexed.

```
let aset : set of CR;
```

Here, a set of elements of type CR (contact record) is defined. Initially, the set is empty. We can add elements to, and remove elements from the set. A remove operation for an element which is not in the set, and an add operation of an element already in the set, have no effect.

```
let indexedset : set {OH} of CR;
```

This statement declares an indexed set of elements of type CR. In fact, this variable stores a set of indices of type OH. For each index, it also stores an element of type CR which can be accessed using the corresponding index.

```
indexedset{obj} := cr;
```

This expression assigns the current value of cr to the indexed set with the index denoted by variable obj of type OH. If there was already an element with that index the corresponding element is changed to the new value of cr. If there was none, the new index including its corresponding element is stored in the variable.

5 Data Structures and Consistency

In this section, we first introduce the basic data structures used to store contact addresses and forwarding pointers. Second, we will define what consistency means for the distributed data stored in contact records.

5.1 Basic data structures

Each node in the search tree maintains a contact record. For simplicity, we assume again that there is only one object handle. Each contact record contains a set of **regional records**, one record for each of the node's children. The regional record stores all data concerning the child node's region. A leaf node has exactly one regional record. The type RCA is used to represent regionalized contact addresses. This leads to the specification shown in Listing 1.

```
type RR is                                              — Regional record
   record
      rcas : set of RCA;                                — Regionalized contact addresses
      ptr : Boolean;                                    — Forwarding pointer to the corresponding child node
   end record;

type CR is                                              — Contact record
   record
      rr : set {NodeID} of RR;                          — Regional records, one per child node
   end record;
```

Listing 1: Basic data structures for storing contact addresses.

For each node there are some variables which hold the essential data for our algorithms. These variables are shown in Listing 2. The listing also includes an initialization function.

```
let parent := "NodeID of parent or NIL";
let children : set of NodeID = "Set of NodeIDs of children";
let cr : CR;                                            — The contact record
let vcr : view series of cr;                            — and its view series

cr := new CR init ⟨∅⟩;
if children = ∅ then
   cr.rr{NIL} := new RR init ⟨∅, false⟩;                — Add one regional record at a leaf node
else
   foreach c ∈ children do
      cr.rr{c} := new RR init ⟨∅, false⟩;               — Add a regional record per child
   od
fi;
```

Listing 2: Basic variables and their initialization.

A contact record is considered **empty** if there are no forwarding pointers and no contact addresses in its regional records. This property can be tested using a boolean function named empty. It returns true if the passed contact record is empty and false otherwise.

5.2 Data consistency

The contents of the contact records and the regional records within the tree have to conform to the following consistency predicates.

Pointer A node stores a forwarding pointer to one of its child nodes in the corresponding regional record if and only if the child node has a non-empty contact record.

Region If there are contact addresses in a regional record then each of them is regionalized in the corresponding region or in one of its subregions.

Exclusion A regional record cannot contain both, a set of contact addresses and a forwarding pointer.

The predicates imply that leaf nodes can store only contact addresses but no forwarding pointers. They also imply that if a node stores a contact address that is regionalized somewhere in a (sub)region of a child node, then there is no data stored neither in this child node nor in one of the child's descendents.

If the entire tree conforms to predicates Pointer, Region, and Exclusion, and the view series of each contact record is empty, then we say the tree is **globally consistent**. The tree is expected to be globally consistent if all operations succeeded and there are no failures. However, during the execution of (concurrent) operations the tree may not be globally consistent.

5.3 Basic structure of update operations

The communication pattern of all update functions is very similar. The initial request is issued by a location resolver at a leaf node. The update function will manipulate the local contact record for the corresponding object handle (e.g., delete or add an address or a forwarding pointer). This is done by appending a view to the view series of the contact record. Then, it may have to forward another request to its parent node (e.g., for establishing or deleting forwarding pointers). After getting a response, the function applies or removes the view that it has appended. Removing a view may be necessary for example if the parent wants to store a contact address in its own node in case of unstable contact addresses.

The only addressee of an RPC is the parent node. Thus, an update function propagates up the tree to a node which does not need to forward the request any further, eventually the root node of the tree. We call the nodes involved in an operation in order of their invocations of functions the **invocation chain**. The highest node in the tree involved in an operation is called the **top node**. A general outline of an update function can be found in Listing 3.

```
operation "update"(...) is
  append view {...} to vcr;                    — Append manipulating view
  call "update" at parent;                                  — Ask parent
  apply view to /remove view from vcr;          — Apply or remove view
  return "return code";                                  — Return response
end "update"
```

Listing 3: General structure of an update function.

In the following sections, we assume that there is no node failure with loss of all or part of the data in a node. However, there may be communication failures like network partitions or temporary broken

network connections. As the **call** primitive blocks until successful invocation, some operations may be blocked for a long time.

6 Inserting a contact address

In this section, we will first outline the behavior of the insert algorithm and then describe the algorithm using the syntax introduced in Section 4.

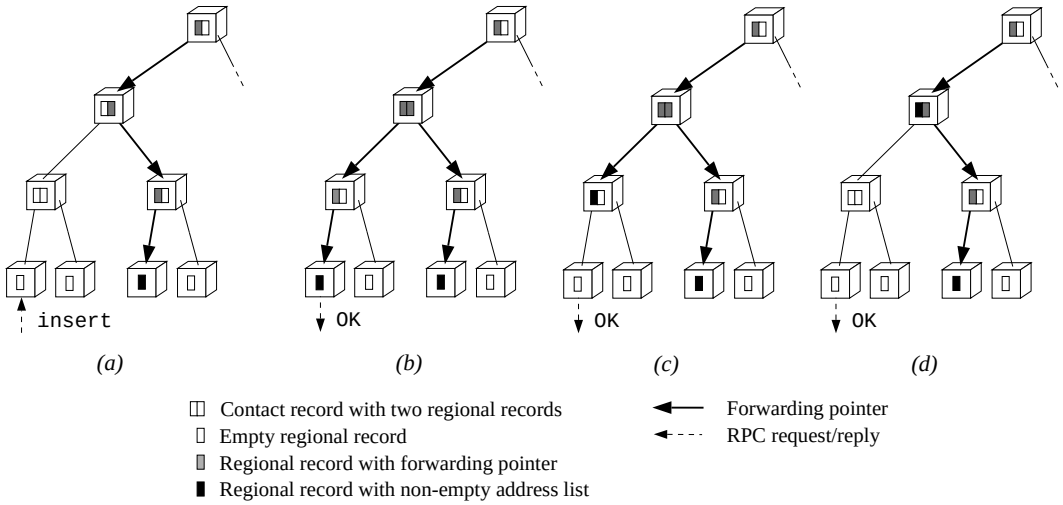

Figure 7: Contact records before and after inserting a new address.

6.1 Outline

Figure 7a shows a partial tree with one contact address that is already registered. The notation is slightly different from that used in Section 3 to distinguish the contents of each regional record. For each child node, a node contains one regional record as mentioned in Section 5.1.

In our example, the insertion of a new contact address takes place at the leftmost leaf node. When the operation has completed the resulting data structure will look like Figure 7b—d depending on where the new address has eventually been stored.

The insert algorithm consist of two functions: insert_rca and insert_chk. At the leaf node, a location resolver always invokes insert_rca. It is the initial request to store a new contact address. If the leaf node has an empty contact record then there is no forwarding pointer from the parent node to this contact record. In this case, insert_rca can decide whether it wants to store the address locally or not. In the latter case, it will invoke insert_rca again, at the parent node.

Figure 8 shows an example. Here, the first invocation of insert_rca decided not to store the address (see Figure 8b). Now, the parent node wants to store the new address, but there is still no forwarding pointer to the local contact record. For installing this forwarding pointer, the parent node invokes the second insert function insert_chk, at the grandparent node. The insert_chk function basically ensures that the missing forwarding pointers will be established (see Figure 8c).

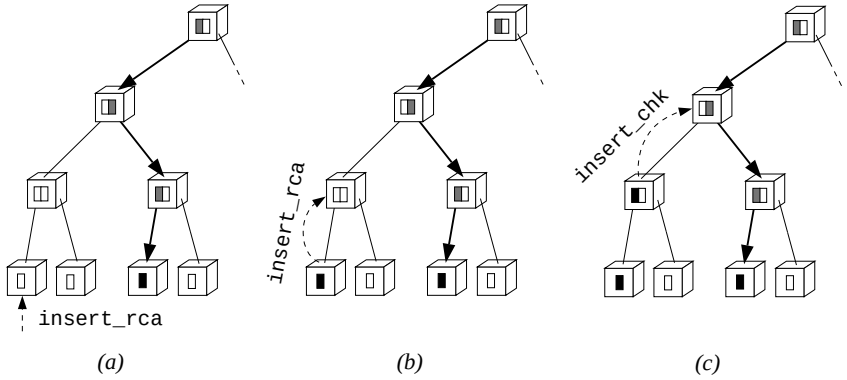

Figure 8: The invocation chain of an insert example.

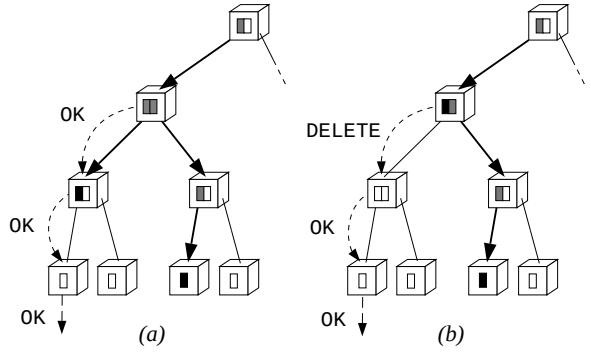

Figure 9: Two possible return paths of the insert example of Figure 8.

Figure 9a shows the result of the usual behavior of insert_chk. As a node that got an insert_chk request may detect that the forwarding pointer was established and deleted often in the recent past it may decide to store the address locally to gain a stable place of storage (see Section 3.2). In this case, insert_chk may return a special response value (DELETE) that tells the child node to delete the new address. This is shown in Figure 9b.

6.2 Algorithm

The insert_rca operation is outlined in Listing 4. The operation gets the new address as an argument. The variable vc is initialized with the current view of the contact record. A view is appended that adds the address to the corresponding regional record. Then, insert_rca checks whether the parent node has to be called. Of course, this is not necessary if the current node is the root of the tree. If the entire contact record is empty the parent node has to be called because there is no forwarding pointer to the local contact record. If the length of the view series of the contact record is nonzero, there are pending operations and the contact record probably contains non-authoritative data. Here, the parent node has to be called, too. This will become clearer from the rest of this section.

If the contact record is empty, the node can decide whether to store the address locally or not. This decision is delegated to the operation shouldstore which returns true if the node should store the address and false if not. Note that shouldstore is effective only in those cases that the contact record is empty.

```
operation insert_rca(rca : RCA) is
  let vc := vcr;
  append view {add rca to self.rr{caller}.rcas} to vcr;
  if parent ≠ NIL ∧ (empty(vc) ∨ views (vc) > 0) then
    if empty(vc) ∧ ¬shouldstore(vcr) then          — Forward request to parent
      call insert_rca(rca) at parent;               — Call parent
      let response := DELETE;                       — Fake response
    else
      let response := call insert_chk(rca) at parent;   — Ask parent
    fi
  else
    let response := OK;                             — Fake response
  fi
  if response = OK then                             — RCA is stored here
    apply view to vcr;
  else — response = DELETE                          — A (grand)parent stored it
    remove view from vcr;
  fi
  return OK;
end insert_rca
```

Listing 4: Insertion of contact addresses.

When an address has already been stored, the decision will always be made to store any succeeding addresses. Only when a record becomes empty, may one of the parent nodes get a chance to store new addresses. But as soon as an address has been stored at some node *S*, all succeeding insertions will take place at *S*.

Regardless of the decision, the node invokes an operation at its parent node. If the contact record was empty and the node does not want to store the address, it forwards an insert_rca request to its parent and fakes a response code DELETE, which will remove the appended view afterwards. In all other cases, the function invokes insert_chk.

If the function does not have to call the parent node it will return a response code OK.

Finally, depending on the response code, the function applies (response code OK) or removes (response code DELETE) the appended view.

The insert_chk function has a similar layout (see Listing 5). It decides whether to append a view that adds the address or to append a view that adds a forwarding pointer. An address is always added if there are already contact addresses in the corresponding regional record. If there are no contact addresses and no forwarding pointer, the node can decide whether to store the new address or not, using the shouldstore operation. The decision may depend on history information about address stability[2]. If the address is not inserted, a forwarding pointer is added.

As in the insert_rca operation, there is a need to check with the parent if the contact record is empty or if the view series contains views. If the node does not send an RPC it returns an OK. On a return code OK it applies the view and returns a code DELETE if it stored the address, and OK if it did not. If it got a return code DELETE from its parent, it removes the view and propagates the parent's return code to the caller.

[2]This information has to be part of the contact record and was not incorporated into the data structures defined in this report.

```
operation insert_chk(rca : RCA) is
  let vc := vcr;
  let rr := vc.rr{caller};
  if rr.rcas ≠ ∅ ∨ (¬rr.ptr ∧ shouldstore(vcr)) then          — RCA should be stored here
    append view {add rca to self.rr{caller}.rcas} to vcr;
    let myresponse := DELETE;
  else                                  — Only a forwarding pointer to the caller should be here
    append view {self.rr{caller}.ptr := true} to vcr;
    let myresponse := OK;
  fi
  if parent ≠ NIL ∧ (empty(vc) ∨ views (vc) > 0) then
    let response := call insert_chk(rca) at parent;                    — Ask parent
  else
    let response := OK;                                                — Fake response
  fi
  if response = OK then                                                — Address is stored here
    apply view to vcr;
    return myresponse;
  else — response = DELETE                                  — A (grand)parent stored the address
    remove view from vcr;
    return DELETE;
  fi
end insert_chk
```

Listing 5: Checking an insert operation with a parent.

6.3 Correctness of the insertion algorithm

We discuss the correctness of the insert algorithm with respect to global consistency. For this discussion, we present some observations.

Observation 1 When an insert function appends a view V, the view which is either removed or applied later by the same invocation of the function, is view V.

Note that a view series only shrinks when a view is removed or applied, and that this always happens after a (possibly faked) response. Consequently, the length of the series at a node equals the number of outstanding calls to a function at the parent node. Note further that both insert functions always call the parent node if there are already outstanding requests to this node. Requests are always responded by the parent in the order of their call. If a function appends a view to a series of length N, then there will be N outstanding calls which will be responded to first, each one reducing the length of the series by one.

Observation 2 The invocation chain ends at the root node, or otherwise at a node with a non-empty contact record whose view series is empty.

The invocation chain ends when no call is made to the parent node. For insert_rca and insert_chk this happens when

$$\text{parent} = \text{NIL} \vee (\neg\text{empty}(\text{vc}) \wedge \textbf{views}\ (\text{vc}) = 0)$$

The first clause states that there is no parent (the current node is the root). The second clause states that the contact record is not empty and there are no views in its associated view series.

Observation 3 When an insert operation has completed, there is exactly one node *S* in the invocation chain where the new address was stored. All nodes above *S* up to and including the top node will store a forwarding pointer. All nodes below *S* down to and including the leaf node will not change their contact records.

The new address is stored by applying a view that adds the address to the local contact record. First, we have to notice that views are applied or removed in reverse order of invocation. The top node applies or removes first, then its client, and so on. This is a direct result from the structure of all update operations (see Listing 3). An operation at a parent is invoked by, and responded to a child, *after* the child has appended a view to its view series, and *before* it applies or removes that view.

Second, we have to verify that if one node applies a view that adds an address, no other node will add that same address as well. Both insert functions can add addresses.

- If insert_rca is called by a location resolver then, trivially, there is no other insert_rca in the invocation chain that can add the address as well. If insert_rca was invoked at a node due to a call from one of its children, the child will fake a response code DELETE which subsequently takes care that the child's previously appended view is removed.
- The insert_chk function may also apply a view that adds an address. In this case, the function always returns a response code DELETE.

We can verify that both insert functions always remove their views if the response code of the call to the parent is DELETE.

Third, we have to check that all nodes above *S* up to and including the top node will add a forwarding pointer. The top node can be a node that got an insert_rca request. In this case, the node will apply a view that adds the new address. Thus, there are no nodes in the invocation chain above node *S*. The top node can also be a node that got an insert_chk request. If this node does not add the address it does add a forwarding pointer. This remains true for all insert_chk calls in the invocation chain until one of the insert_chk functions applies a view to add the address.

6.3.1 Sequential invocations

If there is only one insert operation executed at a time, correctness means that the operation transforms a globally consistent tree into another globally consistent tree with the side effect of inserting a new contact address. The insert operation starts with the invocation of the insert_rca function at a leaf node.

As there is only one insert operation executed at a time, all invocations of insert_rca and insert_chk will never find a contact record with views in its view series. According to Observation 2, the invocation chain ends at the first node which has a non-empty contact record or at the root node. The corresponding regional record can either be empty or can contain RCAs. There could not be a forwarding pointer because that would violate predicate Pointer.

If the regional record at the top node already contains RCAs, the new address is stored there. The insert_rca function will store the address locally if there is a non-empty contact record[3]. This condition includes the case of a non-empty set of addresses in the corresponding regional record. The insert_chk function explicitly adds the address locally if there are already addresses in the regional record.

[3] Note that there are no views in its view series.

If the corresponding regional record is empty, any node on the invocation chain may want to store the address. This decision is made by invoking the shouldstore function. Observation 3 ensures that the resulting data structures are globally consistent.

In all cases, the address is added to a regional record which represents a (super)region of the basic region where the address was inserted. Thus, the resulting data structure is always globally consistent and the address is inserted.

6.3.2 Concurrent invocations at the same leaf node

Now, we consider concurrent invocations of insert operations issued at the same leaf node[4]. Concurrency means here that the next insert operation is already invoked before the previous one has completed. Due to the scheduling policies outlined in Section 4.2, we can observe the following behavior on concurrent invocations:

Observation 4 While a function is blocked on a **call** primitive, the next function may be invoked and scheduled at the same node. A blocked function is resumed only if the response message has arrived and all preceeding functions have completed.

So that we can observe:

Observation 5 The first insert operation of a sequence of operations will not be influenced by the other operations in the sequence.

The first insert operation will thus work correctly as proved in Section 6.3.1.

Observation 6 All insert operations of a set of concurrent invocations at the same leaf node store the new address at the same node *S*.

Combined with Observation 5, this means that the second and all successive invocations store their addresses at the same node as the first one. We will prove Observation 6 in the rest of this subsection. This will prove the correctness of concurrent invocation at the same leaf node.

All invocations, except the first one, will never find an empty contact record at the leaf node. Thus they will never invoke insert_rca at the parent node (see Listing 4). As they will always find at least one view in the record's view series, they will call insert_chk at the parent node. Thus, all invocations, except the first one, start with an insert_rca request, and then have only insert_chk requests in their invocation chain.

The invocation chain of the second operation may be the same as the first one (see Figure 10a). But the invocation chain can also be shorter in the case that in a node *N* the response of a parent call arrives earlier than the request of the second operation (see Figure 10b).

In the first case, it is clear that the second operation will store the address at the same node *S* as the first one, because insert_chk always appends a view that adds an address if there were addresses already stored in the contact record. Likewise, it always appends a view that adds a forwarding pointer if there was already such a pointer. For a third and all following operations this is also true.

[4] Note that we assumed that there is only one object handle. Concurrent operations concerning different object handles in one node never interfere and need no extra consideration.

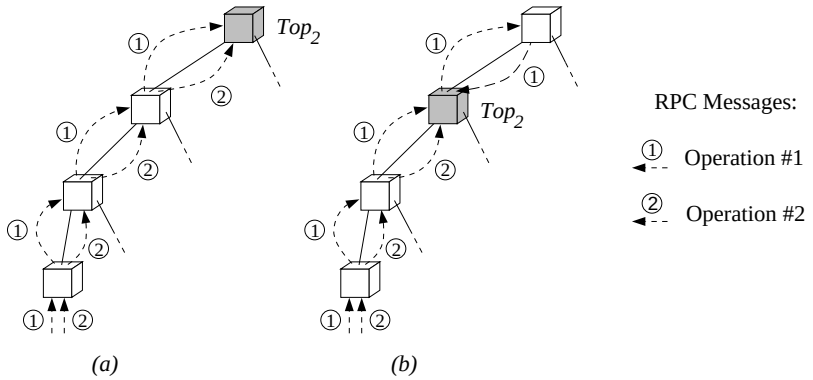

Figure 10: Two concurrent invocations at the same leaf node.

In the second case, a request of the second invocation chain may arrive at node *N* and node *N* has already completely processed the request of the first insert operation. The local contact record has no views in its view series and may be empty, may have a forwarding pointer, or may have a non-empty set of contact addresses. If the record is empty, node *S* is above *N*. The insert_chk invocation of the second operation at node *N* will forward the request until it finds a non-empty contact record at a node (see Observation 2). This node will be node *S*.

If the record contains a forwarding pointer, node *S* is below *N*. The insert_chk function will confirm the forwarding pointer by redundantly applying a view that adds a forwarding pointer at node *N*. In this case node *N* will be the top node of the invocation chain of the second operation. The new address of the second operation will be stored in node *S* following the same reasoning as for the first case.

If the record contains a set of contact addresses, node *N* is node *S* and insert_chk will store the second address there. All nodes below will get a DELETE response.

This proof applies not only to the first and second operation but also for successive operations. If a response of the second operation is already processed in a node that is processing a request of the third operation, then there is also no view in the view series because *all* preceding operations are responded and processed due to the scheduling policy.

6.3.3 Concurrent invocations at different leaf nodes

Concurrent invocations at different leaf nodes do not matter as long as the invocation chains of all operations have no common nodes. If two operations share parts of their invocation chains, we consider them to be concurrent if one of the operations has not completed at the first node that is shared when the second operations invokes a function at that node (see Figure 11).

The correctness proof is similar to the previous one for invocations at the same leaf node. If the first operation stores its address above or at node *N* that is the first shared node between both invocation chains, then we can adopt exactly the same proof.

If the first operation stores its address below node *N* the functions of the second operation will repeat adding forwarding pointers in all nodes above *N*. Then, the functions of the second operation can freely decide where to store the new address (at node *N* or below). This can be done without any concurrency considerations.

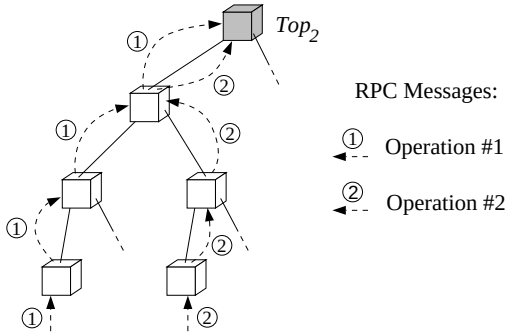

Figure 11: Two concurrent invocations at different leaf nodes.

7 Deleting a contact address

7.1 Algorithm

The delete operation consists of only one function: delete_rca. It is always initially invoked at the same leaf node where a contact address was inserted. If a node cannot make a final decision whether the address is completely deleted or was not found, it will forward the same request to its parent. If a contact record becomes empty because the last address was deleted, the request is also forwarded to the parent to delete the dangling forwarding pointer. This situation can be distinguished by the setting of an additional parameter.

The delete_rca function is outlined in Listing 6. It gets the address to be deleted as an argument. Additionally, it gets a boolean value (parameter ptr) that signals whether a dangling forwarding pointer should be deleted. At a location resolver, delete_rca is always called with ptr set to false.

If the address was found in the local contact record, a view for removing it is appended to the record's view series. If the record now appears to be empty, the parent will have a pointer that needs to be removed. Otherwise, if there were views in the record's view series, the address may possibly be found again at a higher level. In both cases, the parent is called.

In all other cases, the record may appear to be either empty, or not to contain the desired address. In the first case, the request is forwarded because the address may be stored at a higher level. In the other case, the address cannot be found.

7.2 Correctness of the delete algorithm

As with insert, we will first discuss the correctness of sequential delete operation, and then discuss concurrent invocations.

7.2.1 Sequential invocations

If there is only one invocation of delete_rca executed at a time, correctness means that the operation transforms a globally consistent tree into another globally consistent tree if the address was found in the tree. The result is that the address is deleted. If the address cannot be found in the tree, the tree is

```
operation delete_rca(rca : RCA, ptr : Boolean) is
  let vc := vcr;
  if rca ∈ vc.rr{caller}.rcas then                          — Address is there
    append view {remove rca from self.rr{caller}.rcas} to vcr;
    if parent ≠ NIL then
      if empty(vcr) then
        call delete_rca(rca, true) at parent;               — Also delete pointer
      elsif views (vc) > 0 then
        call delete_rca(rca, false) at parent;              — Tentative data: forward req.
      fi
    fi
    apply view to vcr;
    return OK;
  elsif ptr ∧ vc.rr{caller}.ptr then                        — Pointer is there
    append view {self.rr{caller}.ptr := false} to vcr;
    if parent ≠ NIL then
      if empty(vcr) then
        call delete_rca(rca, true) at parent;               — Also delete pointer
      elsif views (vc) > 0 then
        call delete_rca(rca, false) at parent;              — Tentative data: forward req.
      fi
    fi
    apply view to vcr;
    return OK;
  elsif parent ≠ NIL ∧ empty(vcr) then
    return call delete_rca(rca, false) at parent;           — Forward request
  else
    return NOTFOUND;
  fi
end delete_rca
```

Listing 6: Deletion of contact addresses.

not changed and a NOTFOUND response code is returned to the caller.

In the sequential case, the delete function never finds any appended views. Thus, correctness can be easily proved. Let us assume that the address to be deleted is stored at node *S*. Then, we have to prove that the delete operation always has node *S* in its invocation chain, that the address is deleted from *S*'s contact record, and that the consistency criteria remain valid after deletion.

Node *S* is always in the invocation chain of a delete operation, because in a globally consistent tree the address is stored in a node on the path from the leaf node of insertion to the root (predicate Region). Below *S* there can be only empty contact records (predicate Pointer). The delete_rca function forwards the request as long as it has not found a non-empty contact record and will finally arrive at *S*.

If delete_rca finds a non-empty contact record at node *S*, it will remove the address. As the tree is globally consistent this is the only place where the address is stored. The deletion is done using a view. This is immediately applied if the contact record is not made empty by the delete operation.

If the local contact record at *S* became empty, delete_rca is called at the parent with the ptr parameter set to true. This will remove the forwarding pointer from the parent's regional record. This procedure is repeated if the parent's contact record became empty. Thus, the predicate Pointer remains valid. All other predicates are not affected.

If the address is not stored in the tree at all, it is easily seen that the operation will indeed return a NOTFOUND response code.

7.2.2 Concurrent invocations at the same leaf node

Concurrent invocations of delete are much easier to prove correct than concurrent invocations of insert. We have the following observation:

Observation 7 If a tree is globally consistent and if there is a delete operation running that leaves a view in the view series, then the contact record will appear to be empty when evaluating it through the record's associated view series.

A delete operation running on a globally consistent tree will never find appended views. Such a delete operation will immediately apply its own views except when a contact record became empty. In that case, it will always invoke delete_rca with ptr set to true. This will generate views that are also left in the view series (i.e., they are not immediately applied) only if, again, the local contact record became empty.

Having Observation 7 in mind, we can now prove that a second and concurrent delete operation invoked at the same leaf node as the first one will always proceed in the same way as if the first operation had never been invoked. Let *S* be the directory node where all addresses are stored for the region of the leaf node where the delete_rca was initially invoked. Then, if the first operation does not delete the last address in node *S*, that operation will immediately apply its previously appended view, leaving the tree in a globally consistent state.

On the other hand, if the first operation deleted the last address in node *S*, it may have left a view in the record's view series. To the second operation, however, the record will appear to be empty when evaluating it through its view series. Consequently, the second operation is forwarded to the parent node, where it will, again, find an empty contact record (possibly after evaluating it through that record's view

series). Eventually, the root of the tree is reached, resulting in returning a NOTFOUND error, which is propagated down to the leaf node again.

7.2.3 Concurrent invocations at different leaf nodes

In the case of invocations from two different leaf nodes, the invocation chains of two concurrent delete operations will have some nodes in common. One of the operations (named the second one) will eventually find a view that has been left by the other (named the first one) in the view series of the lowest level node the two chains have in common. Denote that node N, and let S again be the node where the addresses from either leaf node are actually stored. Note that N must be at the same or higher level than S, for the only two reasons to append a view is (1) to delete the last address from a contact record (in which case N is the same a S), or (2) to delete a forwarding pointer (in which case N is at a higher level than S).

If N and S are the same nodes, then the contact record found by the second operation will obviously appear to be empty, or otherwise there would not be a view in the record's view series at all. The second operation will never find an address, and thus will never apply or even append a view, because the address for the second operation should also be found at node N, or before that, at a lower level node.

If N and S are not the same node, then N is at a higher level than S. Consequently, the reason why it contained a view in its record's view series must have been caused by a dangling pointer that needed to be removed. In any case, the contact record at node N will appear to be empty to the second operation. But also note that there is no node at a higher level than N that can contain the address the second operation is trying to delete: that address should have been stored at S, which is at a lower level. Consequently, the second operation will eventually return a NOTFOUND error code and never append a view in any node of its invocation chain.

Again, we can easily generalize these arguments to any series of concurrent delete operations.

7.3 Correctness of interfering insert and delete operations

Finally, we come to the point that we prove our algorithms correct when insert and delete operations are executed concurrently. We distinguish invocations at the same leaf node and invocations at different leaf nodes.

7.3.1 Invocations at the same leaf node

Let us first look at a series of requests initiated at a single leaf node. Again, let S be the directory node where the addresses from the leaf node's region are actually stored. Assume the series starts with a number of delete operations. If they do not delete all addresses at S, we can simply ignore their joint effect, because they will not leave any unprocessed views in the view series of the contact record in S.

On the other hand, if all addresses at S are eventually deleted, then the view series at S may contain views that still need to be processed (i.e., applied or removed). However, according to Observation 7, all contact records with such view series will appear to be empty to successive operations. By looking at the code of insert_rca (Listing 4) and insert_chk (Listing 5) we can verify that the behavior of these

functions does not depend on the presence of views as long as the associated contact record appears to be empty. This is expressed in the condition

$$\mathsf{empty}(\mathsf{vc}) \vee \mathbf{views}\ (\mathsf{vc}) > 0$$

We conclude that the effect of a series of delete operations that precede one or more insert operations, can be ignored. Therefore, we can safely consider only series that start with an insert operation. In the following, we first consider a series of concurrent insert requests followed by one delete request. The delete request may find views in the view series of the contact record that have been appended by the preceding insert requests. Consequently, the tree may not be globally consistent.

We first show that the delete operation will find all copies of the address it wants to delete, even if that address is yet to be definitely added to the contact record in S. Note that copies exist in the form of appended views that express that the address is to be (tentatively) added to the present contact record. Delete_rca will always find either an empty contact record whose associated view series contains no views, or otherwise a contact record that appears to be non-empty and of which the view series will certainly contain views appended by the preceding insert operations. In both cases, delete_rca will forward the delete request to the parent, possibly after having appended its own view. It will thus always find all copies of a newly inserted address. Note that this is even true for the case that insert operations have already completed at the top-level nodes of their respective invocation chains (compare this with Figure 10).

We continue with showing that the forwarding pointers will conform to predicate Pointer after the delete operation has completed. If the delete operation does not delete the last address in S's contact record that came from the region where delete_rca was invoked, then there will be no deletion of forwarding pointers above S. However, due to preceeding and pending insert operations, the delete operation may detect dangling forwarding pointers at nodes below S because different nodes may want to store the inserted address while the final decision to store the address in S had not yet been made. In this case, the delete operation will delete the dangling pointer by a corresponding view expression. However, the insert operation will also delete the pointer. The remaining view of the delete operation is harmless and can be neglected.

For the case that the delete operation removes the last address, it will also delete the dangling pointer above S. There can also be views of the previous insert operation that install forwarding pointers. They will be removed by the delete operation because they will be recognized as dangling. This procedure stops at the first node with a non-empty contact record (not considering the dangling pointer). This node is usually the top node of the insert operation.

These proofs can be applied for additional delete operations following the first one. As described before, delete operations do not append views that affect the flow of control in succeeding insert operations. Succeeding delete operations may observe a dangling pointer and handle as just described. We conclude that concurrent invocations of insert and delete operations at the same leaf node behave correctly, because any series of delete operations has no affect on the flow of control in a succeeding insert operation.

7.3.2 Invocations from different leaf nodes

We already showed that concurrent insert operations from different nodes behave correctly. We observe that delete operations do not change anything if they do not find the address to be deleted. Thus,

a delete operation will never interfere with an insert operation that was invoked at a different leaf node. The only exception is that an insert operation may find a view series of nonzero length, to which a concurrent delete operation invoked at a different leaf node, has appended a view to remove a dangling pointer. However, as we stated before, such views do not affect the flow of control of the insert operation as they make the contact record appear to be empty. We conclude that interfering invocations from different leaf nodes behave correctly as well.

8 Looking up contact addresses

We do not go into details of a lookup operation. Lookup operations are fully independent of update operations. They do not have to take into account whether or not a view series is empty, but depend only on viewed values. If a node is unreachable due to a network partition, the lookup operation proceeds as if that node does not exist.

The structure of the update operations guarantees that a lookup at the same leaf node as an update will know about the update. In other words, sequential consistency is guaranteed for operations invoked at the same leaf node. Race conditions may occur for invocations at different leaf nodes, but this is inherent to the distributed nature of the location service.

References

[1] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, N.J., 1991.

[2] P. Homburg, L. van Doorn, M. van Steen, A. Tanenbaum, and W. de Jonge. "An Object Model for Flexible Distributed Systems." In *Proc. 1st ASCI Annual Conference*, pp. 69–78, Heijen, The Netherlands, May 1995.

[3] P. Homburg, M. van Steen, and A.S. Tanenbaum. "An Architecture for A Scalable Wide Area Distributed System." In *Proc. 7th SIGOPS European Workshop*, Connemara, Ireland, Sept. 1996. ACM.

[4] Magnus Sandberg. "An Intermediate Layer to Conduct Experiments in the Globe Location Service." Master's thesis, Vrije Universiteit, Department of Mathematics and Computer Science, Dec. 1996.

[5] M. van Steen, F.J. Hauck, and A.S. Tanenbaum. "A Model for Worldwide Tracking of Distributed Objects." In *Proc. TINA '96*, pp. 203–212, Heidelberg, Germany, Sept. 1996. Eurescom.